UNIT02:SPRITE

【学習要項】
☐Rendering pipeline
☐NDC(Normalized Device Coordinates)
☐Primitive topology
☐Vertices and Input layouts
☐Vertex buffers
☐HLSL
☐Semantics
☐Vertex shader
☐Pixel shader
☐Textures and Data Resource Formats

【演習手順】
1．前回作成した 3dgp.00 プロジェクトを今後継続して使用する
2．シェーダーファイルの追加
①メニュー「プロジェクト」→「新しい項目の追加…」
②「Visual C++」→「HLSL」→「HLSL ヘッダーファイル」→名前：sprite.hlsli

```
1:  struct VS_OUT
2:  {
3:      float4 position : SV_POSITION;
4:      float4 color : COLOR;
5:  };
```

③「Visual C++」→「HLSL」→「頂点シェーダーファイル」→名前：sprite_vs.hlsl

```
1:  #include "sprite.hlsli"
2:  VS_OUT main(float4 position : POSITION, float4 color : COLOR)
3:  {
4:      VS_OUT vout;
5:      vout.position = position;
6:      vout.color = color;
7:      return vout;
8:  }
```

④「Visual C++」→「HLSL」→「ピクセルシェーダーファイル」→名前：sprite_ps.hlsl

```
1:  #include "sprite.hlsli"
2:  float4 main(VS_OUT pin) : SV_TARGET
3:  {
4:      return pin.color;
5:  }
```

⑤ソリューションエクスプローラから sprite_vs.hlsl と sprite_ps.hlsl のプロパティを設定する

「HLSL コンパイラ」→「全般」→「シェーダーモデル」：Shader Model 5.0 (/5_0)
「HLSL コンパイラ」→「出力ファイル」→「オブジェクトファイル名」：%(Filename).cso
「HLSL コンパイラ」→「出力ファイル」→「アセンブリの出力」：アセンブリコードのみ
「HLSL コンパイラ」→「出力ファイル」→「アセンブラー出力ファイル」：%(Filename).cod

⑥sprite_vs.hlsl、sprite_ps.hlsl をコンパイルする
⑦プロジェクトフォルダに sprite_vs.cso、sprite_ps.cso(sprite_vs.cod、sprite_ps.cod)が作成される
※sprite_vs.cod、sprite_ps.cod にはアセンブリコードが書き出されるのでテキストエディタで開いて内容を確認する

⑧ソリューションエクスプローラから 3dgp プロジェクトのプロパティを設定する
※この設定により今後追加するシェーダーファイルのデフォルト値になる
※すべての構成、すべてのプラットフォームで設定する
「HLSL コンパイラ」→「全般」→「シェーダーモデル」：Shader Model 5.0 (/5_0)
「HLSL コンパイラ」→「出力ファイル」→「オブジェクトファイル名」：%(Filename).cso
「HLSL コンパイラ」→「出力ファイル」→「アセンブリの出力」：アセンブリコードのみ
「HLSL コンパイラ」→「出力ファイル」→「アセンブラー出力ファイル」：%(Filename).cod

3．sprite クラスの定義（プロジェクトに sprite.h を新規追加する）

※依存するインクルードファイル

```
#include <d3d11.h>
#include <directxmath.h>
```

①メンバ変数

```
ID3D11VertexShader *vertex_shader;
ID3D11PixelShader *pixel_shader;
ID3D11InputLayout *input_layout;
ID3D11Buffer *vertex_buffer;
```

②メンバ関数

```
void render(ID3D11DeviceContext *immediate_context);
```

③コンストラクタ・デストラクタ

※デストラクタではすべての COM オブジェクトを解放する

```
sprite(ID3D11Device *device);
~sprite();
```

④頂点フォーマット

```
struct vertex
{
  DirectX::XMFLOAT3 position;
  DirectX::XMFLOAT4 color;
};
```

4．sprite クラスのコンストラクタの実装（プロジェクトに sprite.cpp を新規追加する）

※依存するインクルードファイル

```
#include "sprite.h"
#include "misc.h"
#include <sstream>
```

①頂点情報のセット

```
vertex vertices[]
{
  { { -0.5, +0.5, 0 }, { 1, 1, 1, 1 } },
  { { +0.5, +0.5, 0 }, { 1, 0, 0, 1 } },
  { { -0.5, -0.5, 0 }, { 0, 1, 0, 1 } },
  { { +0.5, -0.5, 0 }, { 0, 0, 1, 1 } },
};
```

②頂点バッファオブジェクトの生成

```
 1:  D3D11_BUFFER_DESC buffer_desc{};
 2:  buffer_desc.ByteWidth = sizeof(vertices);
 3:  buffer_desc.Usage = D3D11_USAGE_DEFAULT;
 4:  buffer_desc.BindFlags = D3D11_BIND_VERTEX_BUFFER;
 5:  buffer_desc.CPUAccessFlags = 0;
 6:  buffer_desc.MiscFlags = 0;
 7:  buffer_desc.StructureByteStride = 0;
 8:  D3D11_SUBRESOURCE_DATA subresource_data{};
 9:  subresource_data.pSysMem = vertices;
10:  subresource_data.SysMemPitch = 0;
11:  subresource_data.SysMemSlicePitch = 0;
12:  hr = device->CreateBuffer(&buffer_desc, &subresource_data, &vertex_buffer);
13:  _ASSERT_EXPR(SUCCEEDED(hr), hr_trace(hr));
```

③頂点シェーダーオブジェクトの生成

```
 1:  const char* cso_name{ "sprite_vs.cso" };
 2:
 3:  FILE* fp{};
 4:  fopen_s(&fp, cso_name, "rb");
 5:  _ASSERT_EXPR_A(fp, "CSO File not found");
 6:
 7:  fseek(fp, 0, SEEK_END);
 8:  long cso_sz{ ftell(fp) };
 9:  fseek(fp, 0, SEEK_SET);
10:
11:  std::unique_ptr<unsigned char[]> cso_data{ std::make_unique<unsigned char[]>(cso_sz) };
12:  fread(cso_data.get(), cso_sz, 1, fp);
13:  fclose(fp);
14:
15:  HRESULT hr{ S_OK };
16:  hr = device->CreateVertexShader(cso_data.get(), cso_sz, nullptr, &vertex_shader);
17:  _ASSERT_EXPR(SUCCEEDED(hr), hr_trace(hr));
```

④入力レイアウトオブジェクトの生成

```
 1:  D3D11_INPUT_ELEMENT_DESC input_element_desc[]
 2:  {
 3:      { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0,
 4:        D3D11_APPEND_ALIGNED_ELEMENT, D3D11_INPUT_PER_VERTEX_DATA, 0 },
 5:      { "COLOR", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0,
 6:        D3D11_APPEND_ALIGNED_ELEMENT, D3D11_INPUT_PER_VERTEX_DATA, 0 },
 7:  };
 8:  hr = device->CreateInputLayout(input_element_desc, _countof(input_element_desc),
 9:    cso_data.get(), cso_sz, &input_layout);
10:  _ASSERT_EXPR(SUCCEEDED(hr), hr_trace(hr));
```

⑤ピクセルシェーダーオブジェクトの生成

※CSO ファイル（sprite_ps.cso）のロードは「③頂点シェーダーオブジェクトの生成」を参考にする

```
HRESULT hr = device->CreatePixelShader(cso_data.get(), cso_sz, nullptr, &pixel_shader);
   _ASSERT_EXPR(SUCCEEDED(hr), hr_trace(hr));
```

5．sprite クラスの render メンバ関数の実装

①頂点バッファーのバインド

```
UINT stride{ sizeof(vertex) };
UINT offset{ 0 };
immediate_context->IASetVertexBuffers(0, 1, &vertex_buffer, &stride, &offset);
```

②プリミティブタイプおよびデータの順序に関する情報のバインド

```
immediate_context->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP);
```

③入力レイアウトオブジェクトのバインド

```
immediate_context->IASetInputLayout(input_layout);
```

④シェーダーのバインド

```
immediate_context->VSSetShader(vertex_shader, nullptr, 0);
immediate_context->PSSetShader(pixel_shader, nullptr, 0);
```

⑤プリミティブの描画

```
immediate_context->Draw(4, 0);
```

6．framework クラスのメンバ変数として sprite*型配列を要素数 8 で宣言する

```
sprite* sprites[8];
```

７．framework クラスの initialize メンバ関数で sprite オブジェクトを生成する
※今回は先頭の１個だけを生成する

```
sprites[0] = new sprite(device);
```

８．framework クラスの uninitialize メンバ関数で sprite オブジェクトを解放する

```
for (sprite* p : sprites) delete p;
```

９．framework クラスの render メンバ関数で sprite::render を呼び出す

```
sprites[0]->render(immediate_context);
```

１０．実行し、グラデーションがかかった矩形が表示されることを確認する
１１．矩形を画面全体に表示するために４①の頂点情報の数値を変更し、動作確認する
１２．矩形の描画色（任意）を変更するために４①の頂点情報の数値を変更し、動作確認する
１３．ピクセルシェーダーの main 関数を下記の通りに変更し、動作確認する

```
1:  #include "sprite.hlsli"
2:  float4 main(VS_OUT pin) : SV_TARGET
3:  {
4:    const float2 center = float2(1280 / 2, 720 / 2);
5:    float distance = length(center - pin.position.xy);
6:    if (distance > 200) return 1;
7:    else return float4(1, 0, 0, 1);
8:  }
```

【評価項目】
□矩形の表示
□矩形を画面全体に表示
□矩形の描画色を変更
□日の丸の描画